Pattern Matching for the Masses using Custom Notations

Nic Volanschi

1 rue Michel Jeunet
78300 Poissy, France
tel. +33 130 659 848

Abstract

For many programmers, the notion of "pattern matching" evokes nothing more than regular expressions for matching unstructured text, or technologies such as XPath to match semi-structured data in XML. This common perception of pattern matching is partly due to the success of regular expressions and XPath, which are supported in many popular programming languages today, either as standard libraries or as part of the language. But it is also due to the fact that many programmers never used another elegant form of pattern matching—on structured data, i.e., the native data structures of a programming language. This form of matching is common in functional or logic languages used in research, but unfortunately much less used in the software industry. It is indeed very surprising that none of the popular languages in use today support, in their standard form, a nearly general form of structured data matching, decades after this technology has been discovered and continuously improved.

This paper shows that programmers do not have to wait for next generation languages to integrate pattern matching, neither need they use non-standard pre-processors, thereby losing some advantages that are most important in an industrial setting: official support, compatibility, standardization, etc. Instead, pattern matching of native data in *custom notations* can be implemented as a minimalist library in popular object languages. Thus, some of the comfortable existing notations from logic languages can be reused, existing standard notations for structured data such as JSON

Email address: `nic.volanschi@free.fr` (Nic Volanschi)

(JavaScript Object Notation) can be smoothly extended to support pattern matching, and new notations can be designed.

As in most library implementations of regular expressions, custom notation patterns are simply represented as strings. They can be used in two different modes: interpreted and compiled. This paper presents two open-source implementations of custom matching notations, for Java and JavaScript, exhibiting a reasonable overhead compared to other forms of pattern matching.

Keywords: pattern matching, data notations, customization

1. Introduction

Various programming languages allow pattern matching in suitable notations for particular data structures, such as terms in ML, lists in Prolog, bit strings in Erlang, etc. The long experience of the programming communities of these languages can attest that matching in these notations can be very convenient and may help in writing more elegant programs. In spite of this long-standing and strong evidence, pattern matching has not yet found its way, through mainstream languages, to the wide majority of programmers.

Ideally, any general-purpose programming language, including mainstream languages, should predefine some of these convenient notations for data structures, leveraging the long practice of different existing languages. Additionally, programmers should be able to easily add their own notations for matching both legacy and newly developed data structures, all this without giving up the use of standard tools. Moreover, the notations of various programmers for distinct datatypes should be composable with each other, and also with the predefined notations.

Many solutions have been proposed for extending imperative programming languages such as Java with the pattern matching of general data structures [12, 8, 16, 17]. These fully working systems definitely prove that pattern matching can be integrated within imperative languages, including some of their more advanced features such as: type safety, checks for exhaustiveness or overlapping, and matching optimizations. However, all these unofficial extensions of standard languages are implemented by a pre-processor or a non-standard compiler. Unfortunately, this is a considerable barrier for their widespread adoption in many production projects, because (1) some projects cannot rely on non-standard compiler chains, and (2) pre-processors tend to make debugging more complex. If pattern matching could be implemented without any language extension, even leaving out some of its advanced features mentioned above, this could significantly widen its adoption in the industry. Actually, it has already been proved that pattern matching can be implemented as a pure library (e.g., in Java [27]), but in that approach, patterns are a special kind of native objects, which means that the syntax for patterns is the host language syntax for building objects; as a consequence, pattern syntaxes inspired from other languages, such as Prolog lists or Erlang bit strings, cannot be implemented in this approach.

Another line of work in the literature studies the integration of concrete syntax pattern matching in a host programming language [2, 27]. One of the most general results [26] allows one to integrate an arbitrary context-free "object" language into an arbitrary context-free host programming language. However, this approach is aimed at providing pattern matching for manipulating abstract syntax trees of the object language. Therefore, the concrete syntax approach is extremely useful for designing compilers and program transformation tools, but does not address the matching of arbitrary native objects in a program such as arrays, lists, circular queues, and so on. Besides, this result is also based on building a form of pre-processor that transforms a host+object program into a pure host program.

Thus, none of these two lines of work allow programmers to use pattern matching in concrete syntax on native data objects, while remaining in a 100% standard development environment. Besides, as these approaches do not discuss pattern matching between a string pattern written in an arbitrary syntax, and an arbitrary data object, this notion of string/object pattern matching has yet to be precisely defined.

This paper defines a general concept of matching notations in customized syntaxes. To ensure that such notations are freely composable, we take a pragmatic option which is to parenthesize most sub-patterns. By carefully choosing the parenthesizing technique, existing well-known notations such as JSON (JavaScript Object Notation) [10] and Prolog lists can be smoothly generalized to matching notations, with zero syntactic overhead. By making another pragmatic choice, which is to represent custom patterns as ordinary strings, we show that this concept of matching notations can be very easily implemented in mainstream languages such as Java and JavaScript as a minimalist library, with no extension of the host language. Representing patterns as strings also has the effect of giving up static type checks and static analyses such as pattern exhaustivity or pattern overlapping, but we argue that this shorthand is largely compensated for by providing pattern matching features in mainstream languages in the first place, especially in user-defined syntax.

Matching notations fit particularly well within the object paradigm: a predefined notation can be implemented as a single method at the level of a generic object in the class hierarchy, and can be redefined by more specialized classes, by simply overriding this method.

Pattern matching in custom notations is intended to recognize data items that are native to a programming language, such as integers, strings, structures, objects, lists, etc., and at the same time to decompose them into smaller data items. For instance, the following statement, using one of our pre-defined matching notations, is meant to both recognize a red-black tree *t* having a particular form—a black root and a red left sub-tree—and decompose it into smaller items by binding pattern variables to some sub-items (the left sub-tree's value and own sub-trees, and the root tree's value and right sub-tree)[1]:

```
s = match(t, '{color:"black", left:{color:"red", value:%,
                                    left:%, right:%},
               value:%, right:%}');
if(s) return (s[3] > s[0] + 10)? s[4]: s[2];
```

As can be seen, the retrieved values can then be used in the code.

This predefined notation can be overridden with a custom notation by any programmer. Then, the same data structure might be recognized and decomposed using a completely different pattern, e.g.,

```
s = match(d, "[(% % %) % %]")
```

in which the color of a tree is compactly encoded in the kind of surrounding parentheses, and its components are encoded in a fixed order: left sub-tree, value, and right-subtree.

The main contributions of this paper can be summarized as follows:

- it defines a generic concept of matching notations in custom syntaxes and some important properties such as composability and ambiguity

[1]The patterns in this and other examples have been formatted using whitespace for better readability.

- it shows that this concept can be implemented very concisely, and with reasonable overhead, in two mainstream object languages, thereby bringing not only pattern matching, but matching in custom notations, to most programmers.

This paper is organized as follows. Section 2 defines the notion of matching notations and the related notions of pattern composition, and studies some properties and variations of these. Section 3 shows how the general concept of notations can be implemented very naturally within an object-oriented setting, in two quite different object languages: Java and JavaScript. Section 4 situates the current work in the perspective of other related approaches, and Section 5 presents some ideas of future improvements and concludes by discussing the potential impact of this technology.

2. Matching Notations

Traditionally, pattern matching of structured data is defined between two trees: some data represented by a ground first-order term and a pattern represented as a first-order term including variables. This classic notion of tree matching can be directly used for matching abstract syntax trees (ASTs). For the particular case of concrete syntax pattern matching, the pattern is represented as text, but parsed to an AST; thus, concrete syntax matching is also reduced to the previous classic case of tree matching. Extensions of tree matching exist to deal with matching between two termgraphs [19].

This section defines a notion of pattern matching between a textual pattern, written in some arbitrary syntax, and any kind of data structure in a program. This form of matching:

- defines deterministic composition of different notations nested in the same pattern, which is useful when matching some complex data at once;

- is compatible with data encapsulation, which is essential when integrating it within object-oriented languages;

- allows decomposing the same data in several ways along different patterns, which is needed by pattern languages such as Prolog's list patterns or Erlang's bit string patterns.

In terms of practical advantages, as will be shown in section 3, this definition of matching integrates well with an object-oriented language, by allowing to implement a notation for a class in a single method of that class; in particular, this enables easy overriding of a notation on subclasses. This single method can be implemented either as a straightforward pattern interpreter, or, for more efficiency, as a staged function, taking a pattern and returning a matching function specialized for the given pattern. Finally, no modification of the user class is needed, other than the one-method overriding, so matching behavior can be easily added in legacy programs with minimal intrusion.

2.1. Patterns

The purpose of a matching notation is to recognize and decompose data structures using patterns. As these operations heavily depend on the type of the data structure, a matching notation is always defined in relation to a type $T \subset \mathcal{D}$, where $\mathcal{D}$ represents the set of all data expressible in a programming language.

The most important ingredient of a matching notation is a *pattern language*, that is, a language over a finite alphabet of symbols augmented with a special "variable" symbol: $L_T \subset (\mathcal{A} \uplus \{\%\})^*$, where $\uplus$ denotes the union of disjoint sets. Any word in this language $p \in L_T$ is called a *pattern.* We note $|p| \geq 0$ the number of variables in a pattern p.

The first role of a pattern is to recognize some particular data structures in T. In our framework, this is done by a *filter function* associating each pattern with a subset of the datatype $F_T(p) \subseteq T$, called the *domain* of pattern p. A pattern p is *defined* on data $d \in T$ if $d \in F_T(p)$.

The second role of a pattern is to deconstruct data in T in some of its constituents. In our framework, this is done by a *deconstructor* function D_T associating each pattern with a tuple of projections of the data in T, one for each pattern variable: $D_T(p) = \langle \dots \rho_i, \dots \rangle_{1 \leq i \leq |p|}$, where $\rho_i : F_T(p) \rightarrow T_i$, where $T_i \subset \mathcal{D}$ are other arbitrary types, corresponding to the data components, or "sub-data", in which T can be decomposed using the pattern p. Applying a pattern p to data d means applying the corresponding projections to d, that is, $p(d) = \big\langle \rho_1(d), \dots \rho_{|p|}(d) \big\rangle$.

Summarizing, a *notation* for a type T is a triple $N_T = \langle L_T, F_T, D_T \rangle$, where L_T is a pattern language, F_T is a filter function mapping each pattern to its domain, and D_T is a deconstructor function mapping each pattern to a list of projections.

A notation may be conveniently represented as a set of deconstructor-domain pairs, noted $D_T(p) : F_T(p)$ (read the colon as "defined on"), where the deconstructor is written in "concrete syntax": the pattern p in which any variable has been substituted with the corresponding projection: $p[\rho_i/\%_i]$ (where $\%_i$ denotes the i-th variable in pattern p). For instance, the pattern $p = (\% + \%\mathrm{i})$ together with the deconstructor $D_T(p) = \langle \rho_1, \rho_2 \rangle$ and the domain $F_T(p) = \{d \mid \rho_2(d) \neq 0\}$ can be written in concise form $(\rho_1 + \rho_2\mathrm{i}) : \{d \mid \rho_2(d) \neq 0\}$. If the domain of some pattern is the whole type T, we can simply omit the domain and the preceding colon.

Example 1. *The following is a possible notation for lists:*

- $(\forall n \geq 0)\ \rho_1, \ldots, \rho_n : \{d \mid \mathit{length}(d) = n\}$

where ρ_i represents the projection selecting the i-th element from a list.

Note that the universal quantifier and the ellipsis are not part of the notation, but of the meta-language: they are merely a shorthand for enumerating a set of several patterns. Here, the shorthand represents the patterns $\epsilon : \{d \mid \mathit{length}(d) = 0\}$, where ϵ is also part of the meta-language and represents the empty string, $\rho_1 : \{d \mid \mathit{length}(d) = 1\}$, $\rho_1, \rho_2 : \{d \mid \mathit{length}(d) = 2\}$, and so on. According to the associated domains, the first pattern is defined on empty lists; as it contains no variable, applying it to any empty lists gives the empty tuple. The second pattern, which in its non-substituted form is written %, is defined on lists of one element; its only variable is associated to the projection ρ_1, so its application to a singleton list returns its only element. The third pattern, $\%, \%$, is defined on lists of two elements; its variables are associated to projections ρ_1 and ρ_2 respectively, so its application to a two-element list returns the tuple of those elements, etc. In general, each pattern decomposes a list of a fixed length into the tuple of its elements.

Example 2. *Consider a record datatype for complex numbers, having two floating point fields, "re" and "im". The usual notation from mathematics can be defined very easily on this type as:*

- $\rho_{re} + \rho_{im}\mathbf{i}$
- $\rho_{re} : \{d \mid \rho_{im}(d) = 0\}$

where $\rho_{re}(d) = d.re$ and $\rho_{im}(d) = d.im$.

Note that we note d.re the value of field "re" of the record d. The first pattern, $\%+\%$i, is defined on any complex number, since there is no domain, and decomposes it into its real and imaginary parts, since the first variable is associated to the real projection and the second one to the imaginary projection. The second pattern, %, is only defined on the subset of real numbers, and its application simply returns their value.

Data $d \in T$ is said to *match* a pattern $p \in L_T$ if p is defined on d: $match_T(d, p) \Leftrightarrow p \in L_T \wedge d \in F_T(p)$. The deconstruction of the data obtained by applying the pattern to it, $p(d)$, is called the *result* of the match.

2.2. Composing patterns

Given a pattern $p \in L_T$, a *pattern composition* is an expression of the form $\widehat{p[p_i/\%_i]}_{i \in I \subset [1..|p|]}$, in which some of the variables in p are marked to be substituted with patterns for the corresponding sub-data $p_i \in L_{T_i}$. By performing all the marked substitutions we obtain a *composed pattern*, noted $p[p_i/\%_i]_{i \in I \subset [1..|p|]}$. Thus, we distinguish between the pattern composition expression, which can be thought of as a tree of depth one rooted at p and having p_i as kids, and the resulting composed pattern, which is a word in the language obtained by composing L_T with the languages L_{T_i}. The pattern composition is noted with an extra hat on top, to suggest its tree structure.

The application of a pattern composition to some data d is defined as:

$$\widehat{p[p_i/\%_i]}_{i \in I \subset [1..|p|]}(d) = t_1(d) \cdot \ldots \cdot t_{|p|}(d))$$

where "$\cdot$" denotes tuple concatenation, and:

$$t_i(d) = \begin{cases} \langle \rho_i(d) \rangle \text{ if } i \notin I \\ p_i(\rho_i(d)) \text{ if } i \in I \end{cases}$$

That is, the data is first deconstructed using the top-level pattern, then using the sub-patterns, if any, and all the resulting tuples are concatenated to obtain a single tuple. Note that this definition consistently extends the definition of the application of a flat pattern, because when $I = \emptyset$, i.e., when no composition is done, the former definition reduces to the latter definition.

The match between some data $d \in T$ and a pattern composition can be defined as:

$$match_T(d, \widehat{p[p_i/\%_i]}_{i \in I \subset [1..|p|]}) \Leftrightarrow p \in L_T \wedge d \in F_T(p) \wedge \bigwedge_{i \in I} match_{T_i}(\rho_i(d), p_i)$$

The result of the match is the application of the pattern composition to d.

The application and matching were defined between some data and a pattern composition—a tree of nested patterns. In order to perform matching between some data and a composed pattern—a word in the composed pattern language—, the pattern has to be recognized as the result of a pattern composition, as defined by the following equation:

$$match_T(d,p) \Leftrightarrow p = p'[p_i/\%_i]_{i\in I\subset[1..|p'|]} \wedge match_T(d, \widehat{p'[p_i/\%_i]}_{i\in I}) \qquad (1)$$

and the result of the match is $\widehat{p'[p_i/\%_i]}_{i\in I}(d)$.

Note that according to these definitions, a pattern composition has only one level of pattern nesting; this can be extended to any finite number of nesting in the obvious way. The language obtained by a finite number of compositions starting from L_T is noted $\overline{L_T}$.

2.3. Pattern ambiguity

Actually, equation (1) may admit several solutions, when the composed pattern p may result from several pattern compositions. For instance, if we consider the list notation previously defined and we try to write a pattern for a list of lists of some elements, where the outer list has two elements, and where each of the two inner lists contains exactly one element, we have to express this as the pattern $p = \%, \%$, which results from the composition $p'[\%/\%_1,\ \%/\%_2]$ where $p' = \%, \%$. But this pattern can also represent lists containing a single inner list of length two, that is the result of the composition $p''[\%, \%/\%_1]$ where $p'' = \%$. In other words, the language obtained by composing the above list pattern language with itself is ambiguous.

There is nothing fundamentally wrong with allowing ambiguous pattern languages; matching such an ambiguous pattern may succeed in different ways, producing different results. However, this implies handling several resulting tuples of variable size, which involves a more complicated user interface. Also, recognizers for ambiguous languages may be less efficient than for unambiguous languages. For these reasons, we will impose some constraints for avoiding pattern ambiguity.

One possible way to ensure that pattern composition is unambiguous, is to cast this problem in a classical parsing setting, by requiring for instance that any of the languages L_{T_i} be described by a unambiguous context-free grammar. Once this is assumed, existing results about grammar composition may be used to impose further restrictions on the individual pattern grammars guaranteeing that the resulting composed grammar for language $\overline{L_T}$ is unambiguous.

In the following, we will impose a common and very simple constraint to the individual pattern *languages*, rather than on their grammars; actually, we do not even assume they are generated by a grammar. The constraint is to use parenthesized notations languages, i.e., notations whose patterns always start with an opening parenthesis $(_k \in \mathcal{O}$ and always ends with a corresponding closing parenthesis $)_k \in \mathcal{C}$, where $\mathcal{O} \cap \mathcal{C} = \emptyset$, and $(\mathcal{O} \cup \mathcal{C}) \cap (\mathcal{A} \cup \{\%\}) = \emptyset$. Any pattern language $L_T \subset (\mathcal{A} \uplus \{\%\})^*$ can be trivially transformed to a parenthesized language $L'_T \subset (\mathcal{A} \uplus \{\%\} \uplus \mathcal{O} \uplus \mathcal{C})^*$ by the mapping $w \mapsto (_k.w.)_k$, where the dot denotes word concatenation. By choosing a suitable set of parentheses like "{}[]()", many existing notations are already parenthesized, for instance Prolog lists, Lisp lists, and JSON, to name just a few.

Note that when using only parenthesized languages, words in $\overline{L_T}$ always have parentheses well nested. As a consequence, it is easy to retrieve from a composed pattern the corresponding pattern expression, by simply matching opening and closing parentheses, and back-substituting such sequences by a '%' symbol.

Example 3 (JSON arrays)**.** *If the previous list notation is parenthesized using "[" and "]", we obtain exactly the JSON notation for arrays, which incidentally can also be used for lists of values.*

In this notation, the flat (i.e., non-composed) pattern $p = [\%, \%]$ denotes a list of two elements, the composed pattern [[%],[%]] uniquely denotes $p[\ [\%]/\%_1,\ [\%]/\%_2\ ]$, i.e., a list of two singleton lists; and the pattern [[%,%]] uniquely denotes $p'[\ [\%, \%]/\%_1\ ]$, where $p' = [\%]$, i.e., a singleton list containing a list of length 2. The JSON array pattern [[%],%] matches the array [[1], [2, 3]], yielding the tuple $\langle 1, [2, 3] \rangle$, but does not match the array [[1], 2, 3], because its top-level pattern [%,%] is not defined on arrays of length 3.

2.4. Base notations

With the parenthesizing restriction we imposed, any sub-pattern must be parenthesized. If the notation of a particular sub-data type is already parenthesized, like JSON arrays, this incurs no syntactic overhead for it. However, this particularly penalizes sub-data of atomic base types such as integers, strings, booleans, etc., whose notations are not "naturally" parenthesized. Note that the string notations in most programming languages are not parenthesized in the sense we defined, as we require opening and closing parentheses to be distinct.

Example 4 (JSON objects)**.** *The following is a notation based on the standard JSON notation for objects:*

- $(\forall n \geq 0)(\forall \text{“}key_i\text{”} \in String)_{i \in [1..n]}$
  $\{\text{“}key_1\text{”} : \rho_{key_1}, \ldots \text{“}key_n\text{”} : \rho_{key_n}, \} : \{d \mid (\forall i)\, defined(d.key_i)\}$

where $\rho_{key}(d) = d.key$ represents the projection extracting the value of the field named “key” of an object d, provided that d contains such a field.

Note that the ellipsis belongs to the meta-language: it is just a shorthand for expressing all the patterns containing zero or more pairs of the form $\text{“}key\text{”} : \rho_{key}$, separated by commas.

When composing the JSON object notation above with the usual notations for integers and strings, the pattern $\{\text{“}a\text{”} : (1), \text{“}b\text{”} : (\text{“}two\text{”})\}$ would match an object with at least one field named “a” containing the integer 1 and one field named “b” containing the string “two”. Unfortunately, the notations for the base types must be parenthesized, which deviates from the JSON standard in a quite unpleasant way.

To improve the usability of notations, we can accommodate in our framework, under some conditions, unparenthesized sub-patterns for a set of atomic base types. Consider a set of symbols $\mathcal{B}$ disjoint from $\{\%\} \uplus \mathcal{O} \uplus \mathcal{C}$ representing all possible values of some atomic base types, where each value is represented by a single symbol. This is usually achieved by using a lexical analysis returning a token for each atomic base type. Note that $\mathcal{B}$ is not necessarily disjoint from $\mathcal{A}$.

Now let us examine an example of ambiguous pattern involving an unparenthesized base notation.

Example 5 (Two-way JSON objects)**.** *The following is a matching notation for objects also based on the JSON notation, but adding a second kind of patterns:*

- $(\forall n \geq 0)(\forall \text{“}key_i\text{”} \in String)_{i \in [1..n]}$
  $\{\text{“}key_1\text{”} : \rho_{key_1}, \ldots \text{“}key_n\text{”} : \rho_{key_n}, \} : \{d \mid (\forall i)\, defined(d.key_i)\}$
- $(\forall n \geq 0)(\forall val_i \in \mathcal{B})_{i \in [1..n]}$
  $\{\rho_{val_1} : val_1, \ldots \rho_{val_n} : val_n, \} : \{d \mid (\forall i)(\exists k_i) d.k_i = val_i\}$

where $\rho_{key}(d) = d.key$ is the projection extracting the value of the field "key", and ρ_{val} is the projection extracting the first field name that is associated to the value val, provided the object contains such a field.

Thus, the first kind of patterns extracts the values of some given fields, while the second kind of patterns does the reverse, by extracting the field names that contain some given values.

Given this notation, consider a pattern composed with the unparenthesized notation for strings: {"a" : "b"}. This pattern may be the composition of the pattern {"a" : %} with the string sub-pattern "b", or the composition of the pattern {% : "b"} with the string sub-pattern "a". The former pattern means "extract the value of field 'a' ", while the latter means "extract the first field containing the value 'b' ". Therefore, this composed pattern admits two different top-level patterns, leading to ambiguity. Of course, if we parenthesize the base notations, this ambiguity is removed: the former pattern is expressed as {"a" : ("b")} and checks whether the field "a" contains the value "b", and the second as {("a") : "b"} and checks whether the first field containing the value "b" is exactly the field "a".

More formally, a notation language L_T is *$\mathcal{B}$-ambiguous* if $(\exists p' \neq p'' \in L_T)$ and $(\exists b'_i, b''_j \in \mathcal{B})$ such that:

$$p = p'[b'_i/\%_i]_{i \in I \subset [1..|p'|]} = p''[b''_j/\%_j]_{j \in J \subset [1..|p''|]}$$

In other terms, there is a composed pattern that admits two top-level patterns.

A notation language is *base-ambiguous* if it is $\mathcal{B}$-ambiguous for some set of base symbols $\mathcal{B}$.

Notation languages that are not base-ambiguous can embed unparenthesized base notations, because the top-level pattern is unique for any composed pattern. However, the base-ambiguity property in the above definition may not be easy to check for some notations, because the definition does not give an algorithm to find a "critical pair" of top-level patterns for some given pattern. We therefore give here another equivalent characterization of base-ambiguity that may be easier to check.

Let us note $||w||$ the length of word w, i.e., the number of symbols in w, $Pos(w) = [1..||w||]$ the set of symbol positions of the word w, and $w(k)$ the symbol at position k in word w, where $k \in Pos(w)$.

Two distinct patterns in a notation language are *conflicting* if they have the same length and if at every position, the corresponding symbols are either identical or at least one of them is the variable symbol:

$$||p'|| = ||p''|| \wedge p' \neq p'' \wedge$$
$$(\forall k \in Pos(p'))\ p'(k) = p''(k) \vee p'(k) = \% \vee p''(k) = \%$$

The positions in $Pos(p') = Pos(p'')$ can be partitioned into three disjoint sets:

- $Eq(p', p'') = \{k \in Pos(p') \mid p'(k) = p''(k)\}$
- $Var'(p', p'') = \{k \in Pos(p') \mid p'(k) = \% \wedge p''(k) \neq \%\}$
- $Var''(p', p'') = \{k \in Pos(p') \mid p'(k) \neq \% \wedge p''(k) = \%\}$

The set $Var'(p', p'') \cup Var''(p', p'')$ cannot be empty because this would imply that $p' = p''$. When $Var'(p', p'') = \emptyset$, the pattern p' is *more specific* than p'', because p'' has all the variables in p' plus those in $Var''(p', p'')$, which is not empty. By inverting, when $Var''(p', p'') = \emptyset$, the pattern p'' is more specific than p'. When neither $Var'(p', p'')$ nor $Var''(p', p'')$ are empty, patterns p' and p'' are *overlapping.*

For instance, patterns $p' = \{$“a” $: \%\}$ and $p'' = \{\% :$ “b”$\}$ in the two-way JSON notation for objects are overlapping, $Var'(p', p'') = \{4\}$ and $Var''(p', p'') = \{2\}$.

Proposition 1. *A notation language L_T is base-ambiguous if and only if it contains some pair of conflicting patterns.*

Proof. see Appendix.

Proposition 2. *If a notation language $L_T \subset (\mathcal{A} \uplus \{\%\})^*$ is $\mathcal{B}$-ambiguous, then $\mathcal{A} \cap \mathcal{B} \neq \emptyset$.*

Proof. see Appendix.

Proposition 2 gives thus a necessary condition for a language to be $\mathcal{B}$-ambiguous, given a set of base notations. The result is useful especially taken the other way around: if a base notation set $\mathcal{B}$ is disjoint from some notation's alphabet $\mathcal{A}$ (and this condition is usually trivial to check), then that notation is non-$\mathcal{B}$-ambiguous, so the base notation $\mathcal{B}$ can be embedded unparenthesized in that notation.

Example 6 (Prolog lists). *The Prolog notation for lists is the following:*

- $(\forall n \geq 0)\ [\rho_1, ..., \rho_n] : \{d \mid \mathit{length}(d) = n\}$
- $(\forall n \geq 0)\ [\rho_1, ..., \rho_n | \rho_{>n}] : \{d \mid \mathit{length}(d) \geq n\}$

where ρ_i represents the projection selecting the i-th element from a list, and $\rho_{>n}$ represents the projection returning the list obtained by dropping the n initial elements from a list.

Note that the Prolog notation for lists is a strict generalization of the JSON notation for arrays, in that it can represent not only lists of a fixed length, but also lists of at least a given length; this is the case for the second form of patterns, capturing the tail of the list.

Considering the standard base notations $\mathcal{B}_0$ for integers, strings, and booleans, note that the Prolog list notation is non-$\mathcal{B}_0$-ambiguous by virtue of Proposition 2, simply because $\mathcal{B}_0 \cap \mathcal{A} = \emptyset$, where $\mathcal{A} = \{$"[", ",", "|", "]"$\}$. In particular, the JSON notation for arrays is also non-$\mathcal{B}_0$-ambiguous.

In turn, Proposition 2 cannot help for determining the non-ambiguity of the JSON notation for objects in Example 4, because in this case $\mathcal{A} \cap \mathcal{B}_0 = \mathit{String}$. Indeed, doubly quoted strings can appear both as field names and as base values. Fortunately, we can prove by contradiction that there are no conflicting patterns in the JSON object notation. Assuming that there are two conflicting patterns $\{``\mathit{key}_1" : \%, ...``\mathit{key}_n" : \%\} \neq \{``\mathit{key}'_1" : \%, ...``\mathit{key}'_m" : \%\}$, they must have the same length, so $m = n$; as for all colon-separated pairs, key positions correspond to key positions, and variable positions to variable positions, it follows that $\mathit{key}_i = \mathit{key}'_i$, so the patterns are equal, which is a contradiction. By virtue of Proposition 1, the JSON object notation is non-base-ambiguous, hence it is non-$\mathcal{B}_0$-ambiguous.

Therefore, the JSON object notation, the JSON array notation, and its generalization, the Prolog list notation, are non-$\mathcal{B}_0$-ambiguous, and can thus be composed with the standard unparenthesized base notations.

As an important general observation, it is easy to see that if in a composed notation there is a unique top-level pattern for any composed pattern, further compositions with parenthesized sub-notations keep this property invariant.

In particular, as both the JSON object notation and the JSON array notations are parenthesized—using "{}" and "[]", respectively—, the complete JSON notation, which is an arbitrary composition of these two notations, can be embedded with the unparenthesized standard base notation. For instance, the unique top-level pattern of the pattern {"a":["c":1], "b":"two"} is {“a”:%, “b”:%}. This is also applicable when replacing the JSON array notation with its generalization, the Prolog list notation.

2.5. Named variables

To avoid some degree of complication in the above definitions, we considered patterns with “anonymous” variable positions, all denoted by the ’%’ symbol. Programmers usually find it convenient to associate names with the variable positions. It is straightforward to handle patterns with named variables, noted ‘$\%x$’, where x is the name of the variable, and also non-linear patterns, in which a same variable name can appear several times, standing for the same value. We omit presenting this extension here for space reasons.

2.6. Views and Active patterns

Matching notations can implement a version of views [29] or *active patterns* [25] in user-defined syntax.

Example 7 (Polar view)**.** *Instead of the Cartesian notation for complex numbers defined in Example 2, directly corresponding to their internal representation, a polar notation can be defined as:*

- $< \rho_r : \rho_\phi >$

where $\rho_r(d) = \sqrt{d.re^2 + d.im^2}$ *and* $\rho_\phi(d) = arctan2(d.im, d.re)$*.*

Such a pattern is called active because the data is not simply decomposed into existing slots; rather, the decomposition triggers arbitrarily complex computations that return a different view of the data. It is known that active patterns allow reconciling data abstraction and encapsulation with pattern matching [29].

Indeed, any code using the above polar notation for pattern matching does not have to be changed when the internal implementation of complex number is changed; all what is needed is to change the implementation of this notation, that is, the implementation of pattern filters and of deconstructors.

2.7. Patterns and types

Using our definition of pattern matching, there is no guarantee that a pattern matches data of a single type. Indeed, notations for different types are defined independently of each other, and we do not require their pattern languages to be disjoint.

For instance, the same JSON object notation may be implemented both for objects with fields and for a hash table data structure. In this case, the pattern $\{``a" : 1, ``b" : 2\}$ will match both objects with two fields "a" and "b" containing the given values, and hash tables with two keys "a" and "b" associated to the given values. Thus, the meaning of a pattern exists only in relation to a given notation, which is always associated to a specific type. The recognition role of a pattern works also within a type, to select data *already known to have that type*, that further satisfies some predicate.

That being said, nothing prevents a user to design a closed set of notations for a fixed set of types, where the corresponding pattern languages are all disjoint; in such a closed system, typical for the set of notations built into a programming language, a pattern would also convey type information; however, such closed notation systems are not the bias of this paper, which argues for free composition of independently designed notations.

3. Implementation

Notations, as defined in the previous section, can be implemented very easily in object-oriented languages as a simple library. Indeed, notations and matching are all defined in relation to a type T, that naturally maps to a class in an object language. We show two very concise implementations of matching notations in two radically different object languages:

- JavaScript: an interpreted, dynamically typed language, with prototype-based objects.
- Java: a compiled, statically typed language, with class-based objects and parametric types.

3.1. The JavaScript library

The JavaScript implementation takes the form of a single file called "notations.js", that can be included by other scripts as a library providing pattern matching features in custom notations. The library is available as an open source prototype, and can also be tried on-line in an Internet browser without any installation required [28].

The complete code of the JavaScript library is shown in Figures 1, 2, and 3. This version supports only non-base-ambiguous matching notations, in which base types need not be parenthesized. Thus, we assume that every user-defined notation has been proved non-base-ambiguous before being used, like we did for the predefined JSON object and array notations in the previous section.

A notation is defined on an object type simply by defining for it a method called `matches(pat, off, acc)`, which takes a pattern as an argument. For efficiency reasons, the pattern argument is a string together with an offset into the string; this way, sub-strings of it can be matched without copying them as new patterns. Finally, there is a last input/output argument which is used to accumulate the result of the matching: a tuple, represented as a JavaScript array, containing the variable bindings obtained so far, in left-to-right order. In case of successful match, the method returns an updated offset into the string, which corresponds to the position after the accepted sub-pattern. In case of matching failure, the method raises an exception.

In JavaScript, it is easy to implement a common behavior on all objects, by adding new methods to the top Object class. Any object has a prototype object, from which it inherits the methods and fields; the prototype of any user-defined or primitive object is stored in the "prototype" property of the corresponding constructor. Thus, different `matches` methods can be added to different kinds of objects as shown in Figures 2 and 3 for:

- the base types Number, Boolean and String, implementing the usual notations for these base types
- Array objects, implementing the Prolog notation for lists; recall that this is an extension of the JSON notation for arrays
- any other objects, via the Object's prototype, implementing the JSON notation for objects.

Taken together, these five `matches` methods implement the complete JSON notation, with the slight generalization of Prolog-style lists instead of the simpler JSON lists, which allows one to pattern match variable-sized lists. In order to simplify the presentation, the implementation shown does not tolerate whitespace in the patterns, but this feature can be added easily.

Three worker methods called by the `matches` methods are implemented in Figure 1: one for matching a single character, a second one for matching a token, and a third one for matching a sub-data with a sub-pattern: if the sub-pattern is simply a variable, the corresponding sub-data is pushed on the result array; if the sub-data is null or undefined (these are distinct values in JavaScript), the sub-pattern must match these primitive values directly;

```
function matchChar(ch, pat, off) {
  if(pat.charAt(off) != ch) throw "fail";
  return off + 1;
}

function matchToken(tok, pat, off) {
  if(pat.substr(off, tok.length) != tok) throw "fail";
  return off + tok.length;
}

function matchData(data, pat, off, acc) {
  if(pat.charAt(off) == "%") {
    acc.push(data);
    return off + 1;
  }
  if(data === undefined)
    return matchToken("undefined", pat, off);
  if(data == null)
    return matchToken("null", pat, off);
  return data.matches(pat, off, acc);
}

function match(d, pat) {
  var acc = [], len;
  try {
    len = d.matches(pat, 0, acc);
  } catch(err) {
    if(err == "fail")
      return null;
    else throw err;
  }
  if(len == pat.length)
    return acc;
  else return null;
}
```

Figure 1: The JavaScript implementation.

```
Number.prototype.matches = function(pat, off, acc) {
  return matchToken(this.toString(), pat, off);
};

Boolean.prototype.matches = function(pat, off, acc) {
  return matchToken(this.toString(), pat, off);
};

String.prototype.matches = function(pat, off, acc) {
  if(pat.charAt(off) != "\"") throw "fail";
  off = matchToken(this, pat, off + 1);
  return matchChar("\"", pat, off);
};

Object.prototype.matches = function(pat, off, acc) {
  off = matchChar("{", pat, off);
  if(pat.charAt(off) == "}")
    return off + 1;
  do {
    off = matchChar("\"", pat, off);
    var ix = pat.indexOf("\"", off);
    if(ix >= 0) {
      var field = pat.substring(off, ix);
      if(field in this && pat.charAt(ix + 1) == ":")
        off = matchData(this[field], pat, ix + 2, acc);
    } else throw "fail";
    if(pat.charAt(off) == "}")
      return off + 1;
    off = matchChar(",", pat, off);
  } while(1);
};
```

Figure 2: The predefined base notations and the object notation in JavaScript.

```
Array.prototype.matches = function(pat, off, acc) {
  var ix = 0;
  off = matchChar("[", pat, off);
  if(pat.charAt(off) == "]") {
    if(this.length == 0)
      return off + 1;
    else throw "fail";
  }
  if(ix == this.length) throw "fail";
  off = matchData(this[ix], pat, off, acc);
  ix++;
  while(pat.charAt(off) == ",") {
    off = matchChar(",", pat, off);
    if(ix == this.length) throw "fail";
    off = matchData(this[ix], pat, off, acc);
    ix++;
  }
  if(pat.charAt(off) == "|") {
    off = matchChar("|", pat, off);
    off = matchData(this.slice(ix), pat, off, acc);
  } else if(ix < this.length) throw "fail";
  off = matchChar("]", pat, off);
  return off;
};
```

Figure 3: The predefined array notation in JavaScript.

otherwise, matching between the sub-data and the sub-pattern is delegated to the `matches` method of the sub-data.

The `matches` method of an object type T, when applied to data $d \in T$, implements all the components of a notation defined in Section 2, by acting at the same time:

- as a language recognizer, checking whether the pattern $p \in L_T$,
- as a filter F_T, checking whether $d \in F_T(p)$,
- as a deconstructor D_T, computing the tuple $p(d)$.

The end-user function for matching is the global function `match()` in Figure 1, taking an object and a pattern and returning the resulting array of sub-data or the null value. This global function simply delegates to the object's `matches` method, passing it the whole pattern and an empty accumulator. If the method finishes with no matching failure, it further checks whether the whole pattern was consumed, before reporting a successful match.

Named variables. It is very easy to change this implementation so as to support patterns with named variables, including non-linear patterns, by extending the function `matchData` in Figure 1 to manage mappings from variable names to values. We omit this extension here, though implemented by our open-source prototype.

3.1.1. Using the predefined JSON matching notation

Using this minimalist library, the JavaScript programmer can already match any object using the extended JSON notation, which includes the native notation for base types. For even more convenience, the keys in the JSON notation for objects are accepted without double quotes when the key is a simple identifier; this is consistent with the native notations for objects in JavaScript. For instance, the pattern $\{$"a" : 1, "b" : 2$\}$ can be written simply as $\{a : 1, b : 2\}$. This extension, not shown for brevity, is trivial to implement.

Example 8. *The following function transforms a pair of lists of the form* $\{p{:}l_1,\ q{:}l_2\}$ *into a list of pairs:*

```
function joinPairs(lsts) {
  var s = match(lsts, "{p:[%|%],q:[%|%]}");
  if(!s) return [];
  return [{p:s[0],q:s[2]}].concat(joinPairs({p:s[1],q:s[3]}));
}
```

Note that the value of pattern variable $\%_i$ is retrieved as $s[i-1]$, because array elements in JavaScript start at zero. If the match is successful, the top elements of the two lists are prepended as a pair to the result of the recursive call. Using this function, {p:[1,2], q:[3,4]} is transformed into [{p:1, q:3}, {p:2, q:4}]. If the two lists have different lengths, the unpaired elements are discarded.

This example shows that the native notation in JavaScript integrates gracefully with our JSON patterns, because the latter notation is just an extension of the former; the code is thereby easy to understand and to maintain.

As is standard in pattern matching of structured data, both our definition of matching notations and its implementation support circular data structures. Indeed, the recursion in the definition of matching is related to pattern composition. As any pattern is a composition of a finite number of patterns, the amount of recursion is also finite.

3.2. Defining a custom notation

Programmers can very easily override these default notations for any object type by simply defining a custom `matches` method.

Consider for instance the elegant functional implementation of red-black trees in Haskell, described by Okasaki [18][2]. We can easily obtain a very similar implementation in JavaScript by using a custom, very concise notation for red-black trees.

Red-black trees are binary search trees whose nodes contain values, and are colored black or red, subject to some coloring constraints. They can be implemented as JavaScript objects of the following type Tree. Note how in JavaScript an object type is defined by a constructor with the same name.

```
var red = 0, black = 1;
function Tree(color, left, value, right) {
  this.color = color;
  this.left = left;
```

[2]This example and the next one are inspired by the Matchete paper [8].

```
  this.value = value;
  this.right = right;
}
```

Without pattern matching, the core of the red-black tree implementation, the `balance` method, which re-balances a tree upon the insertion of a new node, has to be coded as shown in Figure 4. As can be seen, the code is very verbose, although much more concise than a standard imperative implementation. This degree of verbosity makes writing such a function tedious to write and error-prone.

Example 9. *To demonstrate the advantage of using custom patterns, let us define the following notation for red-black trees:*

- $[\rho_l\ \rho_v\ \rho_r] : \{d \mid d.color = black\}$
- $(\rho_l\ \rho_v\ \rho_r) : \{d \mid d.color = red\}$

where ρ_l, ρ_v, ρ_r are the projections returning respectively the left sub-tree, the node value, and the right subtree of a given tree. Note how the color of a tree is compactly encoded in the kind of surrounding parentheses.

This notation can be easily implemented by the `matches` method in Figure 5. Using this notation, the `balance` method shown in Figure 4 can be simplified to the one in Figure 6.

Note that pattern matching in this version of the `balance` method is not done using the standard function match() defined previously, but using a disjunctive version of it, which tries a whole list of patterns until the first match. The function matchAny() is simply implemented as:

```
function matchAny(d, plst) {
  for(var i = 0; i < plst.length; i++) {
    var s = match(d, plst[i]);
    if(s) return s;
  }
  return null;
}
```

This example shows that:

- custom notations are very easy to write

```
Tree.prototype.balance = function() {
  if(this.color == black && this.left != null && this.left.left != null &&
     this.left.color == red && this.left.left.color == red)
    return new Tree(red, new Tree(black, this.left.left.left,
                                  this.left.left.value,
                                  this.left.left.right),
                this.left.value,
                new Tree(black, this.left.right, this.value,
                         this.right));
  if(this.color == black && this.left != null &&
     this.left.right != null &&
     this.left.color == red && this.left.right.color == red)
    return new Tree(red, new Tree(black, this.left.left,
                                  this.left.value,
                                  this.left.right.left),
                this.left.right.value,
                new Tree(black, this.left.right.right,
                         this.value, this.right));
  if(this.color == black && this.right != null &&
     this.right.left != null &&
     this.right.color == red && this.right.left.color == red)
    return new Tree(red, new Tree(black, this.left, this.value,
                                  this.right.left.left),
                    this.right.left.value,
                new Tree(black, this.right.left.right, this.right.value,
                         this.right.right));
  if(this.color == black && this.right != null && this.right.right != null &&
     this.right.color == red && this.right.right.color == red)
    return new Tree(red, new Tree(black, this.left, this.value,
                                  this.right.left),
                    this.right.value,
                new Tree(black, this.right.right.left,
                         this.right.right.value,
                         this.right.right.right));
  return this;
};
```

Figure 4: The balance() method without pattern matching

```
Tree.prototype.matches = function(pat, off, acc) {
  off = matchChar((this.color == black?
                "[": "("), pat, off);
  off = matchData(this.left, pat, off, acc);
  off = matchChar(" ", pat, off);
  off = matchData(this.value, pat, off, acc);
  off = matchChar(" ", pat, off);
  off = matchData(this.right, pat, off, acc);
  return matchChar((this.color == black?
                "]": ")"), pat, off);
}
```

Figure 5: The custom notation for red-black trees

```
Tree.prototype.balance = function() {
  var s = matchAny(this, ["[((% % %) % %) % %]",
                          "[(% % (% % %)) % %]",
                          "[% % ((% % %) % %)]",
                          "[% % (% % (% % %))]"]);
  if(!s) return this;
  return new Tree(red, new Tree(black, s[0], s[1], s[2]),
                s[3], new Tree(black, s[4], s[5], s[6]));
};
```

Figure 6: The balance() method using a custom notation.

- due to the first-class status of patterns and of match results, it is easy to implement matching extensions such as disjunctive matching
- pattern matching in a custom notation can make an implementation significantly shorter and less error-prone.

3.3. Compiled notations

The all-in-one `matches` methods described above are a simple and direct implementation of our notion of pattern matching defined in Section 2, but have at least two drawbacks:

- low performance: when matching several data objects with the same pattern, the pattern is traversed again at each match;
- no static checks: when a pattern is invalid with respect to a notation language, this is not reported until a match is first attempted; the `match` function will simply return a null result. This happens also when matching a valid pattern that is not defined on the current data, for instance when matching "[]" with a non-empty list.

Both these drawbacks can be eliminated by separating, in the implementation of a notation, the language recognizer on one hand from the filter and deconstructor on the other hand. In fact, the all-in-one implementations can be seen as *interpreted* notations, while the separated implementation can be seen as *compiled* notations.

Figure 7 shows all the infrastructure code that is needed to support compiled notations, and Figure 8 shows the pre-defined compiled notations for base types. A compiled notation for an object type is implemented by a `matcher` method that acts as a pattern language recognizer and translator: it parses a pattern and, if the pattern is valid, it returns a function.

For instance, the matcher for numbers parses a number; only the case of an integer is depicted. If no number is found, the matcher fails, otherwise it returns a closure comparing some data to the stored sub-pattern. Technically, the `matcher` function also returns an updated offset into the pattern, reflecting the portion that was consumed. The string matcher is similar in spirit; the matcher for booleans is even simpler, as it returns a global function.

Compiling the pattern is implemented by the global function `matcher`, taking a pattern *and a type*. Indeed, the data is not yet available at this time, but the notation to be compiled can be chosen solely based on its type. If

```
function matcher(type, pat) {
  var res = type.prototype.matcher(pat, 0), off = res[0], m = res[1];
  if(off != pat.length) throw "fail";
  return function(d) {
    acc = [];
    try { m(d, acc); } catch(err) {
      if(err = "fail") return null;
      else throw err;
    }
    return acc;
  }
}

function is_undefined(d, acc)
{ if(d !== undefined) throw "fail"; }
function is_null(d, acc)
{ if(d === undefined || d != null) throw "fail"; }

function parseType(type, pat, off) {
  if(pat.charAt(off) == "%") {
    return [off + 1,
            function(data, acc) { acc.push(data); }];
  }
  if(pat.substr(off, 9) == "undefined")
    return [off + 9, is_undefined];
  if(pat.substr(off, 4) == "null")
    return [off + 4, is_null];
  return type.prototype.matcher(pat, off);
}
```

Figure 7: Support for compiled notations in JavaScript.

```
Number.prototype.matcher = function (pat, off) {
  var m = pat.substr(off).match(/^[0-9]+/);
  if(m == null) throw "fail";
  var tok = m[0];
  return [off + tok.length,
          function(d, acc) {
            if(d.toString() != tok) throw "fail";
          }];
};

String.prototype.matcher = function (pat, off) {
  var m = pat.substr(off).match(/^"(([^\\"]|\\.)*)"/);
  if(m == null) throw "fail";
  var tok = m[1];
  return [off + tok.length + 2,
          function(d, acc)
          { if(d != tok) throw "fail"; }];
};

function is_true(d, acc)
{ if(d != true) throw "fail"; }
function is_false(d, acc)
{ if(d != false) throw "fail"; }

Boolean.prototype.matcher = function (pat, off) {
  if(pat.substr(off, 4) == "true")
    return [off + 4, is_true];
  if(pat.substr(off, 5) == "false")
    return [off + 5, is_false];
};
```

Figure 8: Compiled base notations in JavaScript.

the `matcher` method of the indicated type does not fail and consumes all the pattern, it returns a matcher. This matcher is encapsulated by the `matcher` function in a closure that simply calls the stored matcher after initializing its accumulator.

Matching some data with a pattern is done by simply applying to the data the matcher returned by the previous compiling step.

Finally, there is a worker global function called `parseType` that can be used by the matchers of composable notations to compile a sub-pattern corresponding to a sub-data of a given type. In fact, any notation containing variables is composable and must call this function. Function `parseType` also handles the particular cases of null and undefined values. Note that function `parseType` is essentially a staged version of function `matchData` in Figure 1: the latter interprets a pattern with respect to a data, while the former compiles a pattern with respect to a type, and returns a function taking a data and an accumulator to perform a match. Therefore it should be possible to derive function `parseType` by partially evaluating function `matchData`, but this is beyond the scope of this paper.

The red-black trees notation can be implemented in compiled form by the matcher in Figure 9, which calls the worker function `parseType` for both its subtrees and also for its value, and thus builds a closure able to recognize and deconstruct a tree of a given color having two sub-trees of particular shapes. Note that as opposed to the interpreted implementation, the type of values contained in the tree nodes (in our case, a Number) has to be known at parse time, in order to detect incorrectly typed patterns. Of course, the matcher could be parameterized with the type of the values. Figure 9 also shows the `balance` method implemented using the compiled notation for red-black trees. The matcher resulting from the parsing phase is stored in a global variable to ensure that the patterns are compiled only once.

As can be seen from this example, writing and using a compiled notation is hardly more complex than writing and using an interpreted notation.

3.4. The Java library

The Java implementation, also available as an open source prototype [28], is very similar to the JavaScript library described above. Therefore, we only briefly sketch here its main lines and comment on some differences in the implementation caused by differences between the two languages.

First of all, Java is mostly statically typed. That is, inheritance is declared statically, and even if the precise type of an object can be tested dynamically,

```
Tree.prototype.matcher = function(pat, off) {
  var color, fun1, fun2, fun3, res;
  if(pat.charAt(off) == "[") color = black;
  else if(pat.charAt(off) == "(") color = red;
  else throw "fail";
  off++;
  res = parseType(Tree, pat, off); off = res[0]; fun1 = res[1];
  off = matchChar(" ", pat, off);
  res = parseType(Number, pat, off); off = res[0]; fun2 = res[1];
  off = matchChar(" ", pat, off);
  res = parseType(Tree, pat, off); off = res[0]; fun3 = res[1];
  off = matchChar(((color == red)? ")": "]"), pat, off);
  return [off, function(data, acc) {
               if(data == null) throw "fail";
               if(data.color != color) throw "fail";
               fun1(data.left, acc);
               fun2(data.value, acc);
               fun3(data.right, acc);
             }];
}

var m = [matcher(Tree, "[((% % %) % %) % %]"),
         matcher(Tree, "[(% % (% % %)) % %]"),
         matcher(Tree, "[% % ((% % %) % %)]"),
         matcher(Tree, "[% % (% % (% % %))]")];

Tree.prototype.balance = function() {
  s = m[0](this) || m[1](this) || m[2](this) || m[3](this);
  if(!s) return this;
  return new Tree(red, new Tree(black, s[0], s[1], s[2]),
                  s[3], new Tree(black, s[4], s[5], s[6]));
};
```

Figure 9: Compiled notation for red-black trees.

```
public interface Notation {
  boolean matches(String pat, ParsePosition pos, Map sub);
}
public class Matchbox {
  static boolean matchChar(char ch, String pat,
                         ParsePosition pos) {...}
  static boolean matchToken(String token, String pat,
                          ParsePosition pos) {...}
  static boolean matchString(String str, String pat,
                           ParsePosition pos) {...}
  static boolean
    matchMap(Map map, String pat,
             ParsePosition pos, Map sub) {...}
  static boolean matchList(List lst, String pat,
                         ParsePosition pos, Map sub)
  static boolean
    matchObject(Object obj, String pat,
                ParsePosition pos, Map sub) {...}

  public static boolean
    matchData(Object data, String pat, ParsePosition pos,
              Map sub) { ... }
  static boolean matches(Object data, String pat,
                       ParsePosition pos, Map sub) { ... }

  static Map match(Object data, String pat) { ... }
}
```

Figure 10: Interpreted notations in Java (sketch).

```
static boolean matches(Object data, String pat,
                       ParsePosition pos, Map sub) {
  if(data == null)
    return matchToken("null", pat, pos);
  if(data instanceof Matchable)
    return ((Matchable)data).matches(pat, pos, sub);
  if(data instanceof String)
    return matchString((String)data, pat, pos);
  if(data instanceof Number ||
     data instanceof Boolean)
    return matchToken(data.toString(), pat, pos);
  if(data instanceof Map)
    return matchMap((Map)data, pat, pos, sub);
  if(data instanceof List)
    return matchList((List)data, pat, pos, sub);
  // else generic object notation
  return matchObject(data, pat, pos, sub);
}
static Map match(Object data, String pat) {
  HashMap sub = new HashMap();
  ParsePosition pos = new ParsePosition(0);
  if(!matches(data, pat, pos, sub)) return null;
  int len = pos.getIndex();;
  if(len == pat.length()) return sub;
  else return null;
}
```

Figure 11: Interpreted notations in Java (detail).

an object type discipline can be statically enforced, and usually is. Among others, this means that it is not possible to add new matching methods to the top-level Object class, unless the designers of the standard Java library decide to do so in a future version of the language[3]. A first consequence of this is that matching notations must be confined to classes implementing a given interface. This interface, called `Notation`, shown in Figure 10, concerns interpreted matching notations, and requires any implementing class to define a method `matches`[4]. As we noticed a relatively high cost of exception handling in a previous version of our implementation, method `matches` does not signal a matching failure by means of an exception, like the JavaScript implementation, but rather returns a boolean representing the applicability of the pattern to the data. The updated offset is handled by means of a standard object called `ParsePosition`, passed as an argument, that can be both read and updated. The other arguments are the same as in JavaScript: the pattern and an accumulator for the matching result.

Class `Matchbox` defines the end-user matching functions, all static. The main entry point is method `match`, matching an object of an arbitrary type with a pattern. This method, detailed in Figure 11, simply initializes a parse position and an accumulator, and delegates to an appropriate matching method.

As a second consequence of the impossibility to add new methods to system-defined classes, the notations for standard objects such as String must be implemented as static methods such as `matchString`, also defined in class `Matchbox`. Thus, the delegation from method `match` to the appropriate matching method must handle these predefined notations as special cases. This dispatching is implemented in the auxiliary static method `matches`. As can be seen in this method, there are pre-defined notations for the standard classes:

- String: the double-quoted notation,
- List: the JSON array matching notation extended in Prolog-list style,

[3]Some extensions to Java such as eJava [31] have been proposed recently to add methods to predefined classes, but they are not using just the normal Java compiler, in contrast to our full standard compliance.

[4]When using Java version 1.5 or later, substitutions passed as argument *sub* should be represented by a Map<String,Object> instead of simply a Map.

- Map: the JSON matching notation for objects, and
- any other object: also the JSON matching notation for objects.

Indeed, the JSON object notation works as well for the abstract class Map, representing any implementation of a mapping from keys to values, and for any kind of objects that do not have a more appropriate notation, by considering the public fields as keys and their content as values. However, the same JSON object notation must be implemented differently for a Map, where keys and values are accessed using methods such as `Map.containsKey` and `Map.get`, and for a generic Object, where fields and their content are retrieved using the Java reflection API.

Besides these pre-defined notations, class `Matchbox` also defines some worker functions, callable by the pre-defined and user-defined notations. The workers can be used for matching characters, tokens, and some sub-data with a sub-pattern.

Using this simple infrastructure, Java programmers can already use the extended JSON notation for lists, maps, and any other objects containing public fields. They can also easily design a more appropriate custom notation for their classes by simply overriding the `matches` method, as shown in Figure 12 to implement our notations for red-black trees. As can be seen in the `balance` method, the Java code is slightly more verbose than the JavaScript code in Figure 6 because variable values are extracted from a substitution using a `get` method and down-casted to the expected type, which incurs a run-time check.

A compiled notation may also be defined on a class by adding a static method called `matcher`, taking a pattern string and an input/output pattern position and returning a matcher. As a syntax for closures does not currently exist in Java, matchers are defined as objects implementing the following interface:

```
public interface Matcher {
  boolean match(Object data, Map sub);
}
```

3.5. Performance

As matching notations are implemented in a library and not built into the language, their use certainly incurs some overhead. If this performance

```
public class rbTree implements Notation {
  ...
  public boolean matches(String pat,
                         ParsePosition pos,
                         Map sub) {
    return
      Matchbox.matchChar((color == black? '[': '('),
                          pat, pos) &&
      Matchbox.matchData(left, pat, pos, sub) &&
      Matchbox.matchChar(' ', pat, pos) &&
      Matchbox.matchData(value, pat, pos, sub) &&
      Matchbox.matchChar(' ', pat, pos) &&
      Matchbox.matchData(right, pat, pos, sub) &&
      Matchbox.matchChar((color == black? ']': ')'),
                         pat, pos);
  }
  static final String pats[] = {
    "[((% % %) % %) % %]",
    "[(% % (% % %)) % %]",
    "[% % ((% % %) % %)]",
    "[% % (% % (% % %))]"
  };
  rbTree balance() {
    Map s = Matchbox.matchAny(this, pats);
    if(s != null)
      return tree(red,
        tree(black, (rbTree)s.get(0),
             (Integer)s.get(1), (rbTree)s.get(2)),
        (Integer)s.get(3),
        tree(black, (rbTree)s.get(4),
             (Integer)s.get(5), (rbTree)s.get(6)));
    return this;
  }
  static rbTree tree(int c, rbTree l, int v, rbTree r)
  { return new rbTree(c, l, v, r); }
}
```

Figure 12: Defining a custom notation in Java.

price is too high, their usefulness in practice could be seriously compromised, in spite of the added programming comfort, elegance, and maintainability.

Ideally, we would like to perform a fair comparison between the performance of our pattern matching library and that of a built-in pattern matching mechanism, but there is no pattern matching of general data structures built into either Java or JavaScript. Therefore, we can only do a less fair comparison, by measuring the overhead of pattern matching as compared to hand-crafted code performing no pattern matching at all.

Under these constraints, let us consider a somewhat extreme example—that of the implementation of red-black trees described in section 3.2. The insertion into red-black trees (not shown), after creating a new leaf, just calls the method `balance` recursively over the tree. We have already shown three versions of `balance` in JavaScript: one without pattern-matching in Figure 4, a second one using interpreted notations in Figure 6, and a third one using compiled notations in Figure 9. As `balance` applies up to four patterns on each node and does nothing else, we may say that insertion into red-black trees makes a heavy use of pattern matching. We also implemented in Java the three versions of red-black trees insertion, among which only the interpreted notation is shown in Figure 12. Furthermore, we experimented with two versions of the end-user matching functions, accepting only linear patterns, respectively also non-linear patterns.

The benchmarks were executed on a Linux PC equipped with an AMD Athlon XP 2800+ processor running at 2GHz with 256MB of RAM. The execution environment for JavaScript was a Firefox 3.0.18 browser containing a Gecko JavaScript engine. Each timing is the average of 5 different runs. The execution environment for Java was OpenJDK VM and Runtime version 1.6.0.

The JavaScript benchmark inserts 1000 elements in a red-black tree and the Java benchmark inserts 100,000 elements; recall that JavaScript is an interpreted language. The results are given in Table 1. Depending on the version used, the slowdown factor due to pattern matching, when compared with the hand-optimized version, is between 2.3 and 6.1 in Java, and between 3.2 and 6.4 in JavaScript. As expected, compiled notations are almost two times faster than interpreted notations, and as we saw, only slightly more complex to implement; hence, it seems worth investing the extra effort for this performance gain. Also as expected, linear patterns are faster than non-linear patterns, because in the latter case, the engine has to check variable freeness and equality.

Version	**hand-crafted**	**compiled notation**		**interpreted notation**	
		linear	non-linear	linear	non-linear
Java time (ms)	371	854	1444	1631	2257
Java slowdown factor		2.3	3.9	4.4	6.1
JavaScript time (ms)	99	317	365	521	632
JavaScript slowdown factor		3.2	3.7	5.3	6.4

Table 1: Performance of matching notations.

Discussion. While these overheads might seem prohibitive at first sight, it is very instructive to compare them with those of another form of widely used pattern matching: matching of strings using regular expressions. Regular expressions are integrated in the JavaScript language, by means of a standard Regexp object type. Regexp objects are natively parsed in a specific syntax by the interpreter, within slashes /.../, and accepted as arguments to standard methods such as String.match(), as shown in the following JavaScript code sequence, which recognizes messages of the form: "File /usr/lib/mypatterns not readable, please use chmod!!!", and returns the length of the file name:

```
m = s.match(/File ([^ ]*) not (found|readable).*!!!/);
if(m != null) return m[1].length;
```

Regular expressions are also available in Java, provided by the standard library module `java.util.regex`; there is no specific syntax for regular expressions—they are represented as strings, much like our notations. Here is the above code snippet, rephrased in Java:

```
Pattern pattern = Pattern.compile(
          "File ([^ ]*) not (found|readable).*!!!");
Matcher matcher = pattern.matcher(str);
if(matcher.matches()) {
  return matcher.end(1) - matcher.start(1);
}
```

Starting from this relatively naive code sequence, some standard techniques can be applied to improve performance: (1) using non-capturing

Version	**manual**	**optimized regex**	**regex**
Java time (msec)	400	4225	5162
Java slowdown factor		10.6	12.9
JavaScript time (msec)	590	1714	1443
JavaScript slowdown factor		2.9	2.5

Table 2: Performance of regular expressions.

grouping, which avoids saving sub-strings that are not used, such as the "(found|readable)" sub-string, and (2)—for Java only—using possessive quantifiers, which avoid backtrack once a sub-string has been matched; these combined techniques lead to the optimized pattern:

```
File ([^ ]*+) not (?:found|readable).*+!!!
```

For ultimate efficiency, the above code can be further hand-optimized not to use regular expressions at all, by replacing it with carefully written string operations such as extractions and comparisons. For instance the Java hand-crafted version could be:

```
int pos;
if(str.startsWith("File ") && ((pos = str.indexOf(' ', 5)) >= 0)
   && str.startsWith("not ", ++pos) &&
   (str.startsWith("found", (pos += 4)) ||
    str.startsWith("readable", pos)) && str.endsWith("!!!")) {
  return pos - 10;
}
```

The hand-optimized version is less concise, less convenient to write, and much less maintainable, so probably very few Java or JavaScript programmers would ever bother rewriting the code this way, except maybe for a handful of really critical cases. The optimization may be much harder to perform for more complex regular expressions.

Table 2 shows the times in milli-seconds we obtained in Java by matching one million strings, respectively obtained in JavaScript by matching 100,000 strings. Of course, the regular expression compile operation is outside the loop. As it can be seen, using regular expressions in Java incurs a severe overhead—10.6 to 12.9 times slower than the hand-crafted code, and a significant overhead in JavaScript—2.5 to 2.9 slower than the hand-crafted code. Note that the "optimized" regular expression in JavaScript, using non-capturing grouping, is actually slower. It is somewhat natural that the overhead in JavaScript is much lower, as regular expressions are integrated into the language, whereas available as a library in Java. However, in both cases the price to pay for matching is significant.

Note that this regular expression benchmark may be seen as somewhat unfair, as the example does not exploit more powerful aspects of the library such as backtracking. With a more complex pattern, the manually optimized code can be simply too costly to write—in this case, we cannot talk about overhead at all! However, the purpose of this example is just to give an idea about the overhead of regular expression matching on a common case.

Seen in this light, the overhead of matching notations is comparatively very good in Java on the small benchmarks considered here, and may seem acceptable also in JavaScript, at least in cases where the proportion of pattern matching on overall program execution time may be much lighter than in these micro-benchmarks. Like for regular expressions, the overhead of matching notations may be largely justified by the gain in expressiveness, conciseness, maintainability, and reliability—by avoiding error-prone manual optimizations.

4. Related work

Extensions of functional languages related to concrete syntax patterns have been proposed in the past. Aasa *et al.* [1] extended ML with "conctypes" (standing for concrete syntax types), that define an arbitrary context-free syntax, from which are automatically derived: a type definition for the language ASTs, a parser converting language objects into ASTs, and a printer for ASTs. This approach allows one to use patterns both for matching and building ASTs of that language, but is not applicable to defining concrete syntax for native data structures, such as instances of any other ML types, that do not have the stereotyped AST structure. As opposed to that, matching notations allow one to add matching syntax to existing user or pre-defined types, without changing their layout.

Mauny [15] extended ML with a "stream" type using which parsers and printers for user-defined languages may be conveniently written in ML. By integrating such user-defined parsers as hooks into the ML parser itself, it becomes possible to use custom syntax patterns for native data structures. This benefit is similar to our matching notations, but their approach is only applicable to languages with existing pattern-matching support. Indeed, the user patterns are pre-processed as ML ASTs, before being passed to the compiler, so that the matching of user patterns is reduced to the usual ML pattern matching. A further constraint on the applicability of the approach is that the user parser has to call the ML parser for allowing variables or expressions in the patterns. Hence, the ML parser must be accessible to user programs, so the host language implementation must be an open one. In contrast, our approach is applicable to any object language, with no constraint on the host language implementation. Roughly the same comparison applies to a later and more enhanced OCaml extension system, called Camlp4 [20]. Camlp4 completes the support for streams and user-defined parsers with an alternate way for defining parsers using declarative LL(1) grammars. Moreover, these grammars can be extended and redefined. Actually, all the OCaml language is redefined in this formalism; this allows users not only to extend the language, but also to redefine any of its existing constructs. On the other hand, this represents an even more open compiler, in which the whole host language grammar, not just an executable parser, is available to user extensions. User-defined patterns are still reduced to native OCaml patterns, so this approach also requires existing pattern matching support from the host language.

In the same spirit as Campl4, Haskell's quasi-quotation mechanism [13] is implemented as a minimalist patch for the Glasgow Haskell Compiler (GHC). In this system, user-defined parsers do not have access to the whole Haskell grammar, neither can call the standard Haskell parser. Rather, the compiler itself calls user-defined parsers on the customized patterns, and offers them a library for building and manipulating Haskell ASTs. Therefore, this approach also relies on a somewhat more restricted form of open compiler; it also requires pattern matching support from the host language. Another important difference is that when a native datatype in Haskell is given a concrete syntax for matching, the datatype has to be modified to accommodate "anti-quotations", i.e., pattern holes, because user patterns are parsed to this extended datatype. In our approach, custom patterns are directly matched with a native datatype, or directly produce a matcher, so the datatype must not be modified in any way, except for overriding its "matches" method; this may be an important advantage when retro-fitting patterns in a legacy program. Last but not least, Haskell anti-quotations are limited to variable names, because the user parsers cannot call back the Haskell parser on a more complex sub-pattern. This means that different notations cannot be nested in the same pattern. In contrast, nested notation composition is one of the main features of matching notations.

On the other hand, pattern combinators [21] have been used to reimplement pattern matching in functional languages as a pure library, with the added advantage of being able to handle patterns as first-class objects. This is similar to our custom notations, but pattern combinators are furthermore type-safe: providing data of a different type than that expected by a pattern is detected as compile time. This useful feature is obtained by representing patterns as functions, and by building patterns using combinators; thus the types expected by a pattern are exposed to the host language type checker. Another consequence is that combinator-based patterns are written in "abstract syntax", as function applications. As opposed to that, matching notations are written in customizable, concrete syntax, at the price of being type-unsafe.

Views [29] and more recently active patterns [5, 25] extend functional languages to allow matching of a pattern with a custom projection of a data structure. The main goal of views is to allow pattern matching on abstract data types without breaking encapsulation. The pattern is written in a predefined syntax, that of first-order terms. When the root of the term pattern is an active pattern, the matching engine calls a corresponding user-defined function to decompose or pre-process the subject data, and the result is used in further matching. While allowing the same benefit of pattern matching on objects without breaking encapsulation as demonstrated by our example of polar notation for complex numbers, matching notations go beyond views and active patterns by allowing users to customize also the syntax of the projection to be matched. On the other hand, as opposed to views which are bidirectional mappings, matching notations implement no transformation from the projection to the data, and cannot simulate "total" active patterns: they always represent "partial" active patterns [25].

Active patterns have been also added recently to object-oriented languages, for instance in Scala [4] or in Java extensions such as Pizza [17], TOM [16], JMatch [12], and Matchete [8].

Matchete is particularly interesting here for two reasons. Firstly, because it studies the integration of several notations such as first-order terms, array patterns, Erlang-style bit patterns, and regular expressions into Java.

Secondly, because it adapts active patterns to the object way, by extending Java with deconstructors, which are user-defined methods of the object to be matched. This means that the exact deconstructor called during some matching depends on the type of the object, by effect of method dispatching, as in our notations. However, Matchete patterns are written in a limited number of pre-defined syntaxes. Besides those, Matchete also offers user-defined patterns called "extractors". Extractors are user functions taking as arguments some data and a pattern given as a string, so in principle, offer support for patterns written in custom syntax. However, Matchete users have to adopt a non-standard pre-processor. Our approach shows that matching notations can be integrated directly into Java as a library, without any language extension, thereby avoiding the compatibility problems of the preprocessing approach. It is also instructive to see how JSON object patterns can be implemented in both approaches for some user class C. In our custom notations, the whole notation can be implemented as a single method of class C, named `C.match()`, handling different patterns such as $\{a : \%\}$, $\{b : \%\}$, $\{a : \%, b : \%\}$, etc. In Matchete, each of these patterns have to be implemented as a different deconstructor: `C.a~(Object x)`, `C.b~(Object y)`, `C.a_b~(Object x, Object y)`, and so on. Thus, a `match` method of our custom notations may implement an unlimited number of different Matchete deconstructors. Furthermore, if the notation has to be redefined for a subclass of C, only the `match` method has to be redefined in our approach, as opposed to all the different deconstructors in Matchete. From this point of view, one may consider that matching notations achieve a type-unsafe, but more comfortable integration with the object paradigm than Matchete's deconstructors. Note however that a Matchete *extractor* can implement JSON object patterns in a single function, but: the extractor is not a method of the datatype so it cannot be overridden on a sub-class; when using extractors, types are no more checked statically[5]; there is no framework for easily implementing extractors, nor for specializing them for a given pattern, to improve the efficiency; finally, when composing several user notations, these are less integrated syntactically: compare for instance the following nested pattern for a JSON object containing two complex numbers, first written in matching notations, then using Matchete extractors:

- `"{a:(%+%i),b:(%+%i)}"`

[5]As in our implementation, extractors return a tuple of generically-typed Object items.

- `json("{a:%,b:%}")~(complex("%+%i")~(int x, int y),`
  `                    complex("%+%i")~(int s, int t)).`

That is, different matching notations are directly nested into each other, while Matchete extractors are nested as parameterized terms, which has the effect of breaking the pattern into pieces.

JMatch [12] explores a more radical extension of Java, not only supporting a powerful form of pattern matching, but also some logic programming features such as invertible computations. For instance, JMatch can express in a single boolean expression both an object constructor and a de-constructor. Our matching notations are only used for de-constructing data; investigating ways to use them for object construction constitutes an interesting subject for future work. On the other hand, JMatch only supports a fixed notation—that of Java expressions, and is implemented as a pre-processor, not as a library. JMatch also requires learning a new programming style.

Pizza [17] introduces several useful extensions to Java, among which pattern matching of algebraic types. Like in the usual pattern matching of functional languages, patterns are written in a fixed syntax: that of terms; patterns over algebraic types expose the structure of data; users cannot define new patterns, because patterns are in a one-to-one correspondence with case tags. Pizza is also implemented by pre-processing into Java. Interestingly, the ideas from Pizza related to parametric polymorphism strongly influenced the addition of generic types into Java, but Pizza's pattern matching ideas did not have the same destiny. Scala [4] is a newer attempt to incorporate pattern matching features, among many other features, into an object language, not designed as extension of Java, but compiled to Java bytecode and interoperable with Java code and libraries. Scala mainly introduces two constructs for pattern matching in an object setting. Firstly, "case classes" are a means to define union types that can be matched against patterns using type constructors, much the same way as in functional languages. This kind of patterns does not hide the implementation of types. Secondly, "extractors" are a concept very similar to views in functional languages (see above), in that they allow user-defined conversions from one data type to another to be applied implicitly during pattern matching. As such, extractors are a form of extensible matching, and do provide representation independence for objects used in patterns. Both case-classes and extractor patterns are expressed as first-order terms, and this syntax cannot be customized.

TOM [16] also adds pattern matching extensions to different imperative languages, including Java, also using a pre-processor approach. The main goal of TOM is to bring matching and rewriting of tree structures such as ASTs to a wide audience of programmers. In reality, not only trees, but any user datatype can be matched, provided the user implements for it a tree interface, consisting of four functions (get_fun_sym, get_subterm, etc.). List matching is also supported on any data structure provided the user implements for it a list interface, consisting of seven functions. Patterns in TOM are always written using the syntax for terms. In comparison, our notations implement matching of user data in custom syntax, with less overhead (in a single method), and without a pre-processor. On the other hand, TOM patterns are type-safe, checked for exhaustiveness, and the matching is more powerful: it supports equational theories and may compute several matches for a list pattern.

The ability of mixing different notations for data within the same programming system is a striking similarity between matching notations and Intentional Programming (IP) [23]. This ability, in both cases, comes from the fact that notations are not the way data is represented, but just customized projections of the internal representation of data. This allows some degree of ambiguity in the notations, which, in turn, allows mixing notations, and makes notation design lighter than DSL design. However, notations in IP are mainly used for visual rendering, and for wysiwyg-style editing of the data, which can be viewed as a form of manual pattern matching—visually localizing the sub-parts that are manually substituted. As far as we know, automatic matching and transformation operate on the internal tree-structured form, not on its customized projections. As opposed to that, in matching notations, projections are always textual, meant to be included in the program, which allows the program to directly operate with the notations. Thus, with matching notations, programming and domain-specific notations are much closer integrated. More fundamentally, IP proposes a completely new development environment, specifically tailored to support visual projections and editing thereof on the surface side, and to support sophisticated tree transformations on a rich, extensible representation, on the internal side. Matching notations are radically different in the fact that programmers continue to use their favorite language, compiler and their IDE, while custom matching features are added as a minimalist library.

In PADS [7], external data languages may be defined in the form of type definitions annotated with layout information. The type-based description also implicitly defines an in-memory representation of the data, and bidirectional transformations between the internal and external forms: parsers and serializers. It may be thought that this is enough to use PADS for matching any in-memory data in a custom data language. However, while any PADS type is compiled to a native type, it is unclear if any native type can be obtained by compiling a PADS type. Assuming it can, it seems to be no support to match the in-memory representation using the external notation. For instance, in PADS/ML [14], it seems that once the external data is parsed and in-memory, only the built-in pattern matching of ML can be used to match it. In PADX [6], in-memory data can be viewed as XML, by means of a lazy two-way conversion, and matched using XPath, but not using the external notations. Another difference is that programmers have to learn and use an external PADS tool and language in addition to their standard development tools, as opposed to our approach.

In JavaScript the concrete syntaxes for arrays and objects were initially used only to build literal arrays and objects, but starting from version 1.7, they also serve to decompose objects via “de-structuring assignments”, which are a very limited form of pattern matching: they are able to decompose some data according to its structure, but do not return a result (failure or success); the left-hand side can only contain variables, not values; a variable cannot appear more than once; the size of the receiving array must be fixed, etc. Hence, they cannot be used to recognize data having a particular shape. More importantly, the predefined notations for arrays and objects cannot be changed or extended, and new notations cannot be added.

The idea of using plain native objects of a language as patterns, for matching them with other native objects, has been constantly re-discovered in various languages such as Java [27] or JavaScript [24]. The basic technique is to use some specific objects to stand for pattern variables, and to implement pattern matching as tree matching between a native object and a pattern object. This allows implementing a surrogate of term-like pattern matching in a library, and also treating patterns as first class objects, but the syntax of patterns is constrained to be the syntax for representing objects in the host language. This not only means that the syntax is fixed, but also that it is usually not very well-suited for matching purposes.

The OMeta language [30] is somewhat similar in spirit to our work in that it aims at providing flexible pattern matching mechanisms for arbitrary data; its main innovation is to use grammars, annotated with actions, not only for describing and processing text, as is usual, but also for describing and processing structured data. In OMeta’s view, the patterns are the grammar productions, so patterns are written in a BNF-like syntax for expressing grammars, not in a user-defined syntax. A more fundamental difference is that OMeta is an experimental programming language in which actions written in an existing language (Scheme, Smalltalk, etc.) can be embedded; while our goal is adding patterns to an existing language unmodified.

A similar comparison can be done with the Rascal language [11], a recent example of programming language specialized for "language engineering" problems: language definition and parsing, static analysis, program transformation, translation, etc. Rascal offers, among other features, pattern matching of parsed programs in concrete syntax, and rewriting rules relying on this pattern matching support. Using these features, Rascal surely lowers the cost of implementing program manipulation tools. From the very narrow point of view of pattern matching implementation, it can be seen as quite opposite to our custom notations, as it defines a whole new language offering matching in user-defined syntaxes, while our custom notation accommodate custom syntax matching, not limited to parsed programs, into existing languages with no extension at all. Naturally, the trade-offs are very different.

Unparsed patterns introduced in some previous work [22] are similar to interpreted matching notations in that the matching is implemented as a minimalist library and does not involve parsing the patterns. However, unparsed patterns are limited to matching ASTs, that is, tree structures for which a surface syntax is defined. They do not address matching of native data structures such as arrays, neither of circular data. Compiled matching notations are even more different in their implementation technique.

Finally, there have been endless discussions in various programmer communities whether it is worth extending object-oriented languages to support pattern matching of data structures, or rather pattern matching should be rephrased using only standard object mechanism such as method dispatching or multi-dispatching. Our implementation of matching notation shows that pattern matching, even in custom syntaxes, can be implemented using only standard object mechanisms.

4.1. Comparative summary

Among the pattern matching approaches discussed above, the closest to our work are summarized and compared in Table 3, according to several features:

- User-defined syntax: Is it possible to write the patterns in a custom syntax?

	Camlp4	Haskell quasiquote	Pattern Combinators	F# active patterns	Scala	Matchete	JMatch	TOM	MatchO	Notations
User-defined syntax	x	x				x				x
Syntax composition	x									x
Mainstream language						x	x	x	x	x
Any data matchable	x	x	x	x	x	x	x	x	x	x
Extensible matching			x	x	x	x	x	x	x	x
Runtime-type driven						x	x			x
Preserves encapsulation			x	x	x	x	x	x	x	x
Static type checks	x	x	x	x		x	x	x		
Other static checks	x	x		x	x			x		
Non-linear patterns						x	x	x	x	x
First class patterns			x	x					x	x
Constructor patterns	x	x			x		x	x	x	
Parametric patterns			x	x	x	x			x	

Table 3: Pattern matching features — Summary.

Note that in MatchO custom syntaxes can only be used for ASTs, so they are not considered as available for any datatype.

- Syntax composition: Can different custom syntaxes be textually nested in each other?
  Note that in Matchete user-defined syntaxes are not textually nested, although they are part of a same pattern (see the example on page 43).

- Mainstream language: Is the approach integrated in a mainstream language?
  Note that F# and Scala are not considered mainstream languages, even though they are compiled to mainstream platforms, and are therefore inter-operable with mainstream languages. Matchete, JMatch, and TOM are Java extensions, while MatchO and Custom Notations are pure Java (and JavaScript) libraries.

- Any data matchable: Is the approach applicable to match any datatype?
  Note that all the selected systems allow any data to be matched. However, Haskell's quasi-quote requires slightly modifying the datatype in order to be matchable.

- Extensible matching: Can user-defined code be called during matching?
  Note that in Camlp4, user code is called at quotation parsing, not during matching, so it is not considered extensible in the above sense.

- Runtime-type driven: When matching is extensible, is the user-defined code selected depending on the runtime type of the subject data?
  This is the case in Matchete, JMatch, and interpreted custom notations, where the deconstructor is defined as a method of the datatype, and is selected using dynamic dispatching, as opposed to MatchO and Scala extractors, where the matching behavior is a method of the pattern.

- Preserves encapsulation: Is the approach compatible with data encapsulation?
  Note that in Camlp4, quotations are reduced to standard ML patterns, and matched using the standard ML mechanism, which does not allow implementation hiding.

- Static type checks: Are the types of pattern variables checked statically and eventual errors reported at compile time?

- Other static checks: Are other static checks performed statically, such as verifying exhaustiveness or overlap of a set of patterns?
  Note that in Scala, such checks are performed only on case classes, not on extractors.

- Non-linear patterns: Can a variable occur several times in a pattern, standing for the same value?

- First class patterns: Can patterns be used as first-class values, that is, be passed as parameters, returned as results?
  This is the case for F# active patterns and Haskell patterns combinators, where patterns are functions, for MatchO, where patterns are ordinary objects, and for custom notations, where patterns are strings. Note that in Scala, patterns are not values, but first-class status may

be simulated using “pattern-matching anonymous functions”. Similarly, in JMatch passing a pattern as a parameter can be simulated to some extent by passing an object having a pattern method.

- Constructor patterns: Can patterns be used to build objects?
- Parametric patterns: Can patterns be parameterized?
  Note that in Matchete, parameterized pattern exist, but are distinct from deconstructor patterns, which cannot take input parameters. In Scala, type patterns can be parameterized by type variables.

By examining the table, one can see that the matching systems which preserve data encapsulation are exactly those that have an extensible matching engine. This is because all these systems are calling during matching some form of user-defined projectors to decompose the data, thus implementing a concrete view of the data without exposing its implementation. Furthermore, the projectors are in general partial functions, defined on a subset of the datatype. From this point of view, the various systems differ only in the way they map patterns to user-defined projections.

In Haskell pattern combinators and in F# active patterns, a flat pattern *is* a user-defined projection function: in the former case written in continuation-passing style and passing the list of decomposed values to the success continuation, and in the latter case written in direct style and returning the list of decomposed values wrapped in an option type or as a sub-case of an anonymous union type; these constitute partial or total projections.

In Scala, too, a flat extractor pattern is mapped to the unapply method of the corresponding extractor object, returning an optional tuple, econding a partial projection; a case class pattern also corresponds to a partial projection, and a complete set of class patterns to a total projection. Similarly, a flat pattern involving either Matchete deconstructors or JMatch “pattern constructors” is the application of a deconstructor method defined on the subject data; the deconstructor may succeed or fail, thus representing a partial projection. In TOM, a functor name occurring in a pattern is statically mapped to a pair of user-defined functions in the target language (get_fun_sym and get_subterm), and also represents a partial projection.

In our interpreted custom notations, a pattern matching involves a call to the `matches` method of the subject data, which takes the pattern as an argument and produces a list of decomposed values, or fails; this constitutes a partial projection. When a custom notation is compiled, this also produces a partial projection function, to be invoked on the subject data.

Thus, in spite of the apparent diversity of these approaches, the meaning of matching a flat deconstructor pattern[6] with some subject data could be defined for all these systems in the projection-based model introduced in Section 2 if (1) a suitable pattern language and (2) a suitable mapping from patterns to projections are provided for each framework, and (3) the resulting language of composed patterns is proved to be non-ambiguous. In that model, each flat pattern is mapped to a projector and its sub-domain, that is to say, a partial function decomposing the data in a tuple of values. Then the meaning of composed patterns automatically follows from our definitions. This conceptual unification would be useful because currently all these systems define the semantics of matching flat and composed patterns in very different ways, such as: program transformation (e.g., JMatch), informal operational specification (e.g., Matchete), or a concrete implementation (e.g., Pattern combinators).

5. Conclusions

We presented a flexible form of pattern matching, with the following main characteristics: the syntax of patterns can be freely defined; free composability of independent notation is ensured by a simple convention of parenthesizing the patterns; named variables and non-linear patterns can be supported; patterns are first-class and come in two flavors: interpreted and compiled; the implementation is very simple, can be used with no investment, via the predefined notations, and can be enriched very easily with new notations; finally, the performance penalty for using compiled and linear notations ranges from quite tolerable, when compared to that of built-in regular expression matching, to very good, when compared to standard regular expressions libraries.

One consequence of our implementation as a library is that all the pattern variables are dynamically typed. To be precise, in languages that support pattern matching natively (e.g., ML), program variables can directly appear

[6]More work is probably required to express, for instance, the full range of JMatch patterns besides deconstructor patterns, including the use of any Java expression as a pattern, and even in several execution modes, considering different variables as knowns or unknowns.

as pattern variables, and their type can be inferred according to the type of the matched sub-structure. When matching notations are implemented as a library, as shown above, pattern variables are disjoint from program variables, and they always have a generic type (e.g., “object”). In dynamically-typed languages such as JavaScript, this does not make any difference, but in statically-typed languages such as Java, pattern variables have to be downcasted to statically-typed program variables using a runtime check. This point was illustrated by an example in Section 3.4. Another consequence of patterns being implemented as a library is that the compiler cannot perform checks about pattern exhaustiveness or pattern overlap.

Future work may address several aspects. First, the concept of notation, as presented in Section 2 and its specific implementation as a run-time library are not tied to the manual implementation of the pattern language recognizers. Making pattern languages declarative and defining them separately from the program should allow building tools to automatically generate the code for matching methods, still without requiring any language extensions. These external code generators could produce directly bytecode or executable code that would be more efficient than the closures we are currently producing. Another advantage of expressing notations declaratively would be the potential to check them for base-ambiguity, automatically or semi-automatically; one can imagine, for instance, notations that carry a checkable proof of non-base-ambiguity. An interesting question related to declarative notations is whether user-defined notations could be expressed more easily using some form of parser combinators [9]. For that, parser combinators remain to be ported to JavaScript for instance, which does not even provide a recursive let construct. Another difficulty of such an attempt is to keep the overhead of matching reasonably low, at this increased level of abstraction.

Secondly, it would be interesting to explore techniques for reducing the amount of parentheses around the sub-patterns for particular classes of pattern languages; expressing notations more declaratively, as discussed above, could enable using recent advances in parse table composition [3].

Thirdly, notations could be extended to serve not only for de-constructing, but also for constructing data structures in custom syntax. For example, when returning a new red-black tree from function `balance` in Figure 12, the quite verbose nested constructors could be replaced by a single constructor call as follows:

```
if(s != null) return new rbTree("([% % %] % [% % %])", s);
```

We have experimentally written one such a constructor in the Java implementation, and preliminary performance measurements are very encouraging. However, several issues remain to be solved in order to generalize this feature:

- some de-constructing notations do not make sense in a constructive setting; in particular, the default JSON-based notation for de-constructing the public fields of objects of any class does not make sense for constructing objects, as it would violate encapsulation; our formalization should be extended to accommodate such asymmetries

- whenever a same notation can be used both as a constructor and a de-constructor, the user should be able to define both of them via a single method

- the overhead of matching should be kept low, both in the Java and JavaScript implementations, and both for interpreted and compiled notations.

Last but not least, experimenting with many new notations, up to using them in some real-scale applications, could give precious new insights for improving the usability and generality of the approach.

In terms of potential impact, we believe that, beyond bringing more comfort and an elegant mechanism to a wide audience of programmers, this creativity-enabling technology can promote programmers from their current role of passive language *consumers* to that of active language *contributors*. A rich eco-system of notation designers and users, crossing the traditional boundaries between programming languages, may change the shape of data notations in languages to come.

Today, the language evolution cycle is extremely long: the inspiration of most language designers is rooted in past programming languages, combined with recent research results and prototypes, to design the next generation of languages. As data notations are currently built into languages, they evolve at the same slow pace. Also, the clear separation between designers and users makes that the former cannot be exposed to all of the many, and extremely different, needs that the latter encounter in real-scale projects all over the world. As opposed to that, in the open evolution model enabled by custom notations, language designers can take their inspiration from a vast pool of different notations, mostly coming from real practice; they can generalize and select the most elegant and powerful ones for pushing into the next version of a language.

Acknowledgements

Many thanks to the anonymous reviewers, whose constructive criticism helped in tremendously improving the paper. This work was not funded by any organization, but some contributions are worth mentioning. The public transportation companies of the Paris area supported the review of related work by providing seats for reading virtually all of the cited papers. The heavy traffic conditions in Hanoi encouraged the author's work during holidays, and the beauties of the Cat Ba island provided inspiration for comparing related approaches. None of these forms of support would have been enough without Nathalie's permanent understanding. Thanks also to Christian Rinderknecht for feedback on an early version of the paper, and to my parents for repeated encouragements. This paper is dedicated to the memory of Mihai Zamfirescu Sr., who viewed research as a tree growing activity.

References

[1] A. Aasa, K. Petersson, D. Synek. "Concrete syntax for data objects in functional languages." In LISP and functional programming (LFP), July 25–27, 1988.

[2] M. Bravenboer, E. Visser. 2004. "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions." In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 24–28, 2004.

[3] M. Bravenboer, E. Visser. "Parse Table Composition." In Software Language Engineering (SLE), September 29–30, 2008. LNCS 5452 (2009), pp. 74–94.

[4] B. Emir, M. Odersky, J. Williams. "Matching Objects with Patterns." In European Conference on Object-Oriented Programming (ECOOP), July 30–August 3, 2007. LNCS 4609 (2007), pp. 273–298.

[5] M. Erwig. "Active patterns." In Implementation of Functional Languages (IFL), September 16–18, 1996. LNCS 1268 (1997), pp. 21–40.

[6] M. Fernandez, K. Fisher, R. Gruber, Y. Mandelbaum. "PADX: Querying Large-scale Ad Hoc Data with XQuery." In PLAN-X, January 14, 2006.

[7] K. Fisher, R. Gruber. "PADS: A Domain-Specific Language for Processing Ad Hoc Data." In Programming Language Design and Implementation (PLDI), June 11–15, 2005. SIGPLAN Not. 40 (2005) pp. 295–304.

[8] M. Hirzel, N. Nystrom, B. Bloom, J. Vitek. "Matchete: Paths through the Pattern Matching Jungle." In Practical Aspects of Declarative Languages (PADL), January 7–8, 2008. LNCS 4902 (2008), pp. 150–166.

[9] G. Hutton. "Higher-order Functions for Parsing." In Journal of Functional Programming, 2 (1992) pp. 323–343.

[10] JSON (JavaScript Object Notation). http://www.json.org/. 2010.

[11] P. Klint, T. van der Storm, J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation." In Source Code Analysis and Manipulation (SCAM), September 20–21, 2009, pp. 168–177.

[12] J. Liu, A. Myers. "JMatch: Java plus Pattern Matching." Technical Report 2002-1878, Computer Science Dept., Cornell University. October 2002, revised April 2005.

[13] G. Mainland. "Why It's Nice to be Quoted: Quasiquoting for Haskell." In ACM SIGPLAN workshop on Haskell, September 30, 2007.

[14] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, A. Gleyzer. "PADS/ML: A Functional Data Description Language." In Principles of programming languages (POPL), January 17–19, 2007. SIGPLAN Not. 42 (2007) pp. 77–83.

[15] M. Mauny. "Parsers and printers as stream destructors and constructors embedded in functional languages." In Functional programming languages and computer architecture (FPCA), September 11–13, 1989, pp. 360–370.

[16] P.-E. Moreau, C. Ringeissen, M. Vittek. "A Pattern Matching Compiler for Multiple Target Languages." In Compiler Construction (CC), April 5–13, 2003. LNCS 2622 (2003), pp. 61–76.

[17] M. Odersky, P. Wadler. "Pizza into Java: translating theory into practice." In Principles of programming languages (POPL), January 15–17, 1997, pp. 146–159.

[18] C. Okasaki. “Red-black trees in a functional setting.” Journal of Functional Programming, 9 (1999) pp. 471–477.

[19] D. Plump. “Term Graph Rewriting”. In Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools, chapter 1, pages 3-61, eds. H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg. World Scientific, 1999.

[20] D. de Rauglaudre. “Camlp4 - Tutorial. version 3.07.” http://caml.inria.fr/pub/docs/tutorial-camlp4/. 2003.

[21] M. Rhiger. “Type-safe pattern combinators.” Journal of Functional Programming, 19 (2009) pp. 145–156.

[22] C. Rinderknecht, N. Volanschi. “Theory and Practice of Unparsed Patterns for Metacompilation.” Science of Computer Programming, 75 (2010) pp. 85–105.

[23] C. Simonyi, M. Christerson, S. Clifford. “Intentional software.” In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 22–26, 2006.

[24] B. Stein. “Functional Pattern Matching in JavaScript.” http://www.bramstein.com/projects/jfun/. 2010.

[25] D. Syme, G. Neverov, J. Margetson. “Extensible pattern matching via a lightweight language extension.” In International Conference on Functional Programming (ICFP), October 01–03, 2007.

[26] E. Visser. “Meta-programming with Concrete Object Syntax.” In Generative Programming and Component Engineering (GPCE), October 6–8, 2002. LNCS 2487 (2002), pp. 299–315.

[27] J. Visser. “Matching objects without language extension.” Journal of Object Technology, 5 (2006) pp. 81–100.

[28] N. Volanschi. “myPatterns: The open source implementations of custom notations.” http://mypatterns.free.fr/. 2010.

[29] P. Wadler. “Views: a way for pattern matching to cohabit with data abstraction.” In Principles of Programming Languages (POPL). January 21–23, 1987.

[30] A. Warth, I. Piumarta. “OMeta: an object-oriented language for pattern matching.” In Symposium on Dynamic languages (DLS), October 22, 2007.

[31] A. Warth, M. Stanojevic, T. Millstein. “Statically Scoped Object Adaptation with Expanders.” In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 22–26, 2006.

Appendix

Proof of Proposition 1

Proof. To prove the forward implication, consider a base-ambiguous language L_T. This means that for some $\mathcal{B}$ there exist two patterns $p' \neq p'' \in L_T$ and some base values $(b'_i)_{i \in I \subset [1..|p'|]}, (b''_j)_{j \in J \subset [1..|p''|]} \in \mathcal{B}$ such that

$$p'[b'_i/\%_i]_{i \in I \subset [1..|p'|]} = p''[b''_j/\%_j]_{j \in J \subset [1..|p''|]} = p$$

Note that as each variable symbol is replaced by one symbol in $\mathcal{B}$, p, p', and p'' have all the same length, so $Pos(p) = Pos(p') = Pos(p'') = [1..||p||]$. Let Pos_I and Pos_J be the positions of the variables $\%_i$ in p' and $\%_j$ in p'' respectively. Then:

- Any position in $Pos(p) \setminus (Pos_I \cup Pos_J)$ was not substituted, neither in p' nor in p'', and therefore corresponds to the same symbol from $\mathcal{A} \uplus \{\%\}$ in all p, p', and p''.
- Any position in $Pos_J \cup Pos_I$ corresponds to some variable $\%_i$ in p', or to some variable $\%_j$ in p'', or both.

Thus, patterns p' and p'' are conflicting, because all the three conditions in the definition are satisfied: the patterns are distinct, they have the same length and at every position, the corresponding symbols are either identical or at least one of them is the variable symbol. This proves the forward implication.

To prove the backward implication, consider a notation language L_T containing two conflicting patterns $p' \neq p''$. We can build a pattern p that can have both p' and p'' as top-level patterns, as follows:

$$p \triangleq p'[p''(k)/\%_{i_k}]_{k \in Var'(p',p'')}$$

where $p'(k) = \%_{i_k}$; there must be such an $i_k \in [1..|p'|]$ because $k \in Var'(p', p'')$, so $p'(k) = \%$.

That is, p is built by substituting in p' all variables that do not exist in p'' at the same position k by exactly the symbol that p'' has at that position, $p''(k)$. It can be easily shown that the following equality also holds:

$$p = p''[p'(k)/\%_{j_k}]_{k \in Var''(p',p'')}$$

for some $j_k \in [1..|p''|]$, by case reasoning:

1. For any position $k_0 \in Eq(p', p'')$ we have

$$\begin{aligned} p(k_0) &= p'[p''(k)/\%_{i_k}]_{k \in Var'(p',p'')}(k_0) = p'(k_0) \\ &= p''(k_0) = p''[p'(k)/\%_{j_k}]_{k \in Var''(p',p'')}(k_0) \end{aligned}$$

   where the second and last equality hold because $k_0 \notin Var'(p', p'') \cup Var''(p', p'')$.
2. For any position $k_1 \in Var'(p', p'')$ we have

$$\begin{aligned} p(k_1) &= p'[p''(k)/\%_{i_k}]_{k \in Var'(p',p'')}(k_1) = p''(k_1) \\ &= p''[p'(k)/\%_{j_k}]_{k \in Var''(p',p'')}(k_1) \end{aligned}$$

   where the last equality holds because $k_1 \notin Var''(p', p'')$
3. For any position $k_2 \in Var''(p', p'')$ we have

$$\begin{aligned} p(k_2) &= p'[p''(k)/\%_{i_k}]_{k \in Var'(p',p'')}(k_2) = p'(k_2) \\ &= p''[p'(k)/\%_{j_k}]_{k \in Var''(p',p'')}(k_2) \end{aligned}$$

   where the second equality holds because $k_2 \notin Var'(p', p'')$

Therefore, by choosing for example the set of base symbols $\mathcal{B} = \mathcal{A}$, which allows to substitute variables in p' with symbols in p'' and vice-versa, the pattern p shows that the notation language L_T is $\mathcal{B}$-ambiguous. Hence, L_T is base-ambiguous. $\square$

Proof of Proposition 2

Proof. As the language is $\mathcal{B}$-ambiguous, there are two patterns $p' \neq p''$ both instantiated to p using base values $b'_i, b''_j \in \mathcal{B}$:

$$p = p'[b'_i/\%_i]_{i \in I \subset [1..|p'|]} = p''[b''_j/\%_j]_{j \in J \subset [1..|p''|]}$$

By virtue of Proposition 1, p' and p'' are conflicting. The sets $Var'(p', p'')$ and $Var''(p', p'')$ cannot be both empty—otherwise it would imply that $p' = p''$. If, for instance, the first is non-empty (the other case is symmetric), then there is a position $k \in Var'(p', p'')$, which means that $(\exists i_k \in I)$ such that $p'(k) = \%_{i_k} \wedge p''(k) \neq \%$. So

$$b_{i_k} = p'[b'_i/\%_i]_{i \in I \subset [1..|p'|]}(k) = p''[b''_j/\%_j]_{j \in J \subset [1..|p''|]}(k) = p''(k)$$

But the word $p'' \in L_T \subset (\mathcal{A} \uplus \{\%\})^*$, and we saw that $p''(k) \neq \%$, so $p''(k) \in \mathcal{A} \cap \mathcal{B}$. $\square$